Administración de memoria

El concepto de multiprogramación en los sistemas operativos nos lleva a un problema de administración de memoria, ya que la administración de procesos necesita precisamente de una buena administración de memoria. La memoria es un recurso que es particionado y asignado a los procesos que están listos, de esta manera tanto el procesador como la memoria son utilizados de manera eficiente. En general se particiona en dos partes: una para el sistema operativo y otra para el área de usuario. Además, el área de usuario necesita ser dividida en múltiples partes para sus procesos. Dividir la memoria para los diversos procesos que se alojarán en ella necesita una administración apropiada, la cual, incluye asignación y protección. Para entender los principios de la administración de memoria son necesarios los siguientes conceptos.

Asignación estática y dinámica

En general la asignación de memoria se realiza de manera estática y de manera dinámica. **La estática** se realiza antes de la ejecución de un proceso y hay dos formas en las que se puede llevar a cabo.

1. Cuando la ubicación del proceso en la memoria se conoce en tiempo de compilación, el compilador genera un código para ese proceso. Si el proceso necesita ser cambiado dentro de la memoria, entonces el código debe ser recompilado.
2. Cuando la ubicación del proceso no se conoce en tiempo de compilación, el compilador no genera una dirección de memoria actual, sino que genera un código reubicable, es decir, la dirección relativa sobre algún punto conocido. Dentro del código reubicable es fácil cargar el proceso en una ubicación distinta (de ser necesario) y no es necesario compilar el código otra vez. En ambos métodos de asignación estática se debe conocer el tamaño antes de comenzar la ejecución del proceso.

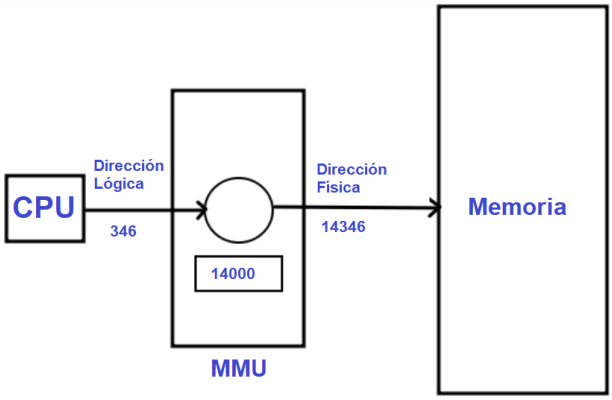
Algunos de los primeros lenguajes de programación, como FORTRAN, utilizan la asignación estática únicamente. Aunque es la forma más sencilla de asignar memoria hay algunas desventajas: el tamaño de los objetos debe ser conocido en tiempo de compilación por lo tanto no se pueden cambiar a menos que se vuelva a compilar. Por otro lado, al no poder cambiar el tamaño de los objetos podríamos estar desperdiciando espacio si creamos estructuras más grandes de lo necesario. Finalmente, no se pueden implementar procedimientos recursivos.

La asignación de memoria dinámica se lleva a cabo hasta que el proceso comienza su ejecución. En esta el tamaño de los datos, durante el tiempo de ejecución, puede ser variable, así que se asigna conforme se va necesitando. Por lo tanto, es posible asignar el tamaño de cada estructura de manera precisa. Recordemos que cuando el sistema operativo carga un programa en memoria para ejecutarlo le asigna cuatro partes lógicas en la memoria principal: texto, datos (estáticos), pila ($stack$) y una zona libre llamada heap, la cual, va a contener los datos dinámicos.

Dirección lógica y dirección física

Una dirección identifica de manera única una ubicación de memoria, y en general tenemos dos tipos: la dirección lógica es una dirección virtual que puede ser vista por el usuario. Por otro lado, la dirección física no puede ser vista directamente por el usuario. Además, la dirección lógica es usada como una referencia para acceder a la dirección física. La principal diferencia entre estas es que la dirección lógica es generada por el CPU durante la ejecución de un programa mientras que la dirección física hace referencia a una ubicación dentro de la unidad de memoria. En la siguiente tabla se presentan otras diferencias entre estos dos tipos de memoria.

Dirección lógica	**Dirección Física**
Dirección virtual generada por el CPU.	Es una ubicación dentro de la unidad de memoria.
El espacio de direcciones lógico es un conjunto de direcciones lógicas creadas por el CPU en referencia a un programa.	El espacio de direcciones físicas es el conjunto de direcciones mapeadas desde el espacio de direcciones lógico.
El usuario puede ver la dirección lógica de un programa.	El usuario no puede ver la dirección física.
El usuario usa la dirección lógica para acceder a la dirección física.	El usuario no puede acceder directamente a la dirección física.
Es generada por el CPU	Es calculada por la unidad de gestión de memoria (MMU – Memory Management Unit).



Swapping

Hay situaciones en multiprogramación donde no hay memoria para ejecutar un nuevo proceso. En tal caso sacar un proceso de la memoria para que haya espacio para un nuevo proceso podría parecer una buena solución, sin embargo:

- ¿Dónde pondríamos el proceso que fue sacado de la memoria?
- De todos los procesos que residen en memoria ¿cuál deberá salir para darle cabida a otro?
- ¿En qué parte de la memoria será alojado el proceso una vez que sea devuelto?

En general, los procesos que son sacados de la memoria son almacenados en un dispositivo de almacenamiento secundario (en la mayoría de los casos es el disco duro). La acción de sacar un proceso de la memoria es llamado: $swap - out$. Por otro lado, la acción de devolver un proceso a la memoria es llamado swap-in. El espacio para intercambio $(swap\ space)$ deberá ser lo suficientemente grande para que se puedan alojar los procesos sacados de la memoria.

Así cada vez que el planificador $(scheduler)$ seleccione un proceso para ser ejecutado, el despachador $(dispatcher)$ verifica si el proceso deseado está en la cola de listos. Si no se encuentra ahí entonces un proceso es sacado de la memoria $(swap - out)$ y el proceso elegido por el planificador es devuelto $swap - in)$ a la memoria.

En relación con la pregunta sobre cuál proceso deberá salir de la memoria hay diferentes métodos para determinarlo:

- En la planificación $round\ Robin$ los procesos son ejecutados de acuerdo con su tiempo $quantum$. Si dicho tiempo expira y el proceso no ha terminado su ejecución podrá salir de la memoria, es decir, aplicar $swap - out$ sobre dicho proceso. El $quantum$ es el tiempo máximo que un proceso puede hacer uso del procesado. Este puede ser fijo o variable y además puede ser el mismo valor para todos los procesos o ser distinto.
- En la planificación guiada por prioridad, si se quiere que un proceso con alta prioridad sea ejecutado, entonces un proceso con baja prioridad será sacado de la memoria.

Fragmentación. Cuando se asigna una partición a un proceso, puede ser que su tamaño sea menor al de la partición, dejando un espacio que no podrá ser ocupado por otro proceso después de la asignación. Este desperdicio de memoria, interno a una partición, se conoce como $fragmentación\ interna$. Por otro lado, mientras se asigna o se retira memoria a un proceso a través de diversos métodos, es posible que queden pequeños espacios en varias particiones a lo largo de la memoria, de tal manera que, si se combinan, pueden satisfacer los requerimientos de otro proceso. Sin embargo, esos espacios no pueden ser combinados. Este espacio total de memoria fragmentada, externa a todas las particiones se conoce como $fragmentación\ externa$.

Asignación de memoria contigua

La asignación de memoria contigua fue implementada particionando la memoria en varias regiones. La partición se puede realizar usando memoria fija o variable. En estos métodos de asignación contigua un proceso se ubica en su totalidad en posiciones consecutivas de memoria. Por lo tanto, se debe buscar y asignar aquella partición de memoria que se ajuste al proceso. Cabe aclarar que aquella partición de memoria que aún no ha sido asignada se conoce como $agujero\ (hole)$. Por lo tanto, se busca un agujero apropiado para que se aloje el proceso correspondiente. Cuando el proceso termina, la memoria ocupada

se libera y ese agujero vuelve a estar disponible. Cuando todos los agujeros están asignados a un conjunto de procesos, el resto de los procesos entran a un estado de espera. Tan pronto como un proceso termina su ejecución y se libera un agujero, este es asignado a un proceso en espera.

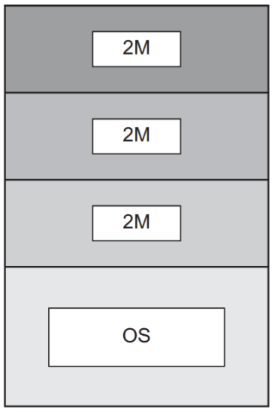
Asignación contigua con particiones fijas

En este método una vez que se asignan las particiones ya no pueden cambiarse. Para asignar memoria a los procesos en las particiones el SO crea una tabla para almacenar la información de las particiones en la memoria: $tabla\ de\ descripción\ de\ particiones\ (PDT)$.

ID de partición	Dirección de inicio	Tamaño	Estatus de asignación

En este método, el planificador a largo plazo es el que realiza la programación de tareas y decide cuál proceso debe llevarse a la memoria. Después averigua el tamaño del proceso que se va a cargar y solicita al administrador de memoria que asigne un agujero en la memoria. El administrador de memoria utiliza una técnica de asignación, como las que veremos más adelante, para encontrar el agujero que mejor se ajuste al proceso. Después de que se asigna el agujero, el planificador coloca el proceso en dicha partición.

Aquí cabe preguntar cuál debería ser el tamaño de las particiones de memoria. Una manera de hacerlo es tener particiones del mismo tamaño. Aunque esto tiene sus desventajas porque un proceso puede ser muy grande para ajustarse a una partición. Otra desventaja es que el proceso sea muy pequeño, lo cual, puede llevar a la fragmentación de la memoria.

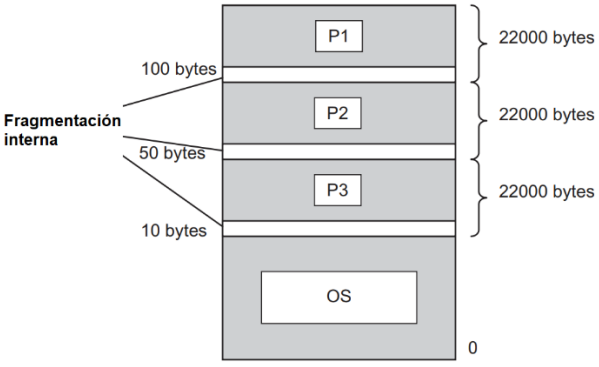


Ejemplo. Tenemos tres procesos $p_1, p_2\ y\ p_3$ de $21900, 21950\ y\ 21990\ bytes$ de tamaño respectivamente, y además se necesita memoria. Si tenemos particiones de tamaño fijo de $22000\ bytes$ entonces habrá una fragmentación interna.

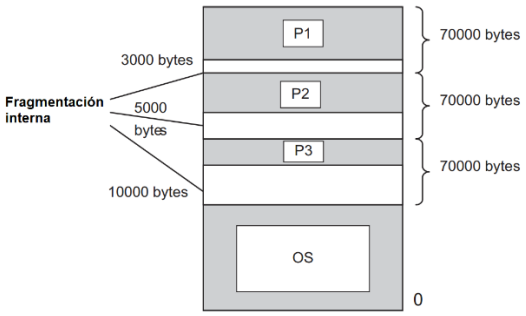
Espacio sobrante en la primera partición $= 22000 - 21900 = 100\ bytes$

Espacio sobrante en la segunda partición $= 22000 - 21950 = 50\ bytes$

Espacio sobrante en la tercera partición = $22000 - 21990 = 10\ bytes$

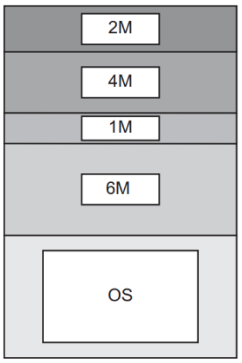


Ejemplo. Tenemos tres procesos $p_1, p_2\ y\ p_3$ de $67000, 65000\ y\ 60000\ bytes$ respectivamente y se necesita espacio para estos. Si tenemos particiones de memoria de tamaño $70000\ bytes$ y alojamos estos procesos en particiones contiguas ¿cómo será la fragmentación? En la siguiente imagen se puede ver que incluso el espacio desperdiciado puede ser más grande que en anterior ejemplo.



Como podemos ver al asignar particiones contiguas de tamaño fijo habrá un desperdicio de un espacio considerable de memoria (si sumamos todos los espacios sobrantes).

Una alternativa es usar particiones de diferente tamaño en la memoria. De esta forma se puede acomodar procesos de menor a mayor tamaño según corresponda, desperdiciando así menor memoria.



Al usar particiones de diferente tamaño se mejora el rendimiento al asignar memoria a los procesos. Sin embargo, este método (actualmente obsoleto) tiene las siguientes desventajas:

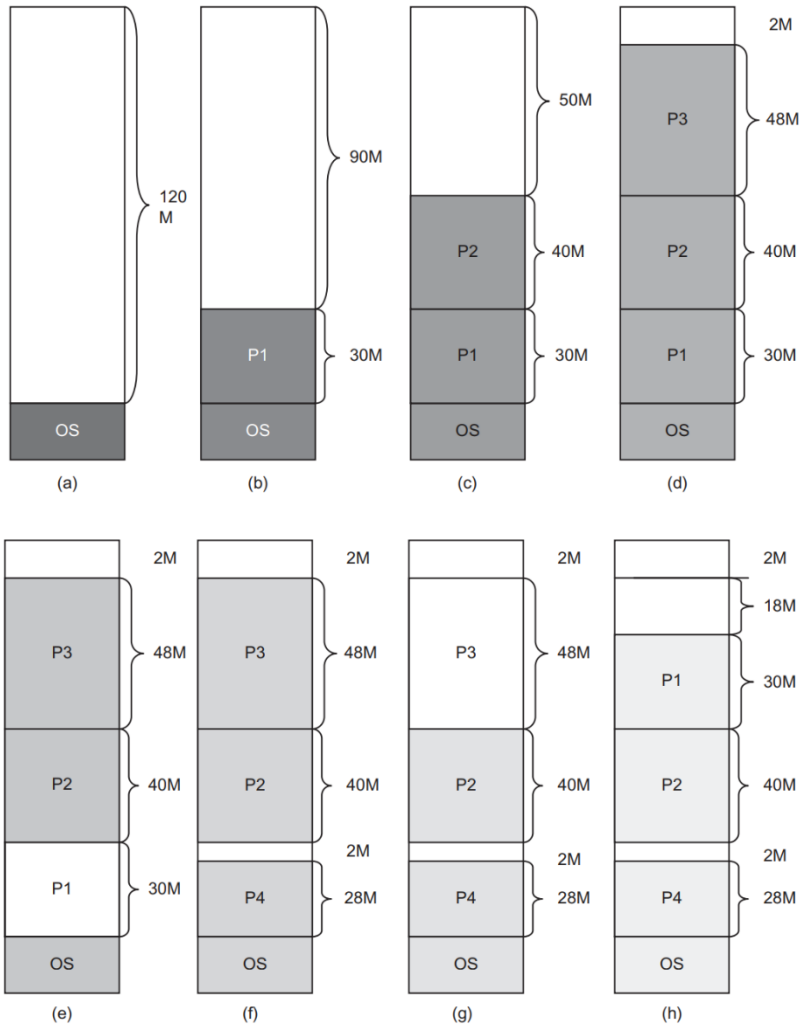
- Puede haber algunos procesos cuyos requerimientos de memoria no se conoce por adelantado. Así que en este caso incluso el uso de particiones de diferente tamaño causará fragmentación interna. Debido a esta fragmentación, puede haber espacios de memoria dispersos en varias particiones. Si esos espacios fueran combinados sería posible asignárselo a un proceso. Sin embargo, en un método de particionamiento fijo no es posible. Así que también resulta en un problema de fragmentación externa. Por lo tanto, el método de particionamiento fijo sufre de ambos tipos de fragmentación.

Ejemplo. Tres procesos $p_1, p_2\ y\ p_3$ de $19900, 19990\ y\ 19888\ bytes$ de tamaño respectivamente necesitan espacio en memoria. Si tenemos particiones de igual tamaño, de $20000\ bytes$, y alojamos a los procesos en dichas particiones podremos observar que las particiones dejarán $100\ bytes, 10\ bytes$ y $112\ bytes$ respectivamente a los procesos $p_1, p_2\ y\ p_3$. Esto lleva a una partición interna de $222\ bytes$. De manera que si un proceso de $200\ bytes$ de tamaño necesita memoria no podría ser acomodado, a pesar de que el tamaño de la memoria libre es más grande que el proceso. Esto se debe a que dichos espacios libres no son contiguos, esto tiene como resultado una fragmentación externa.

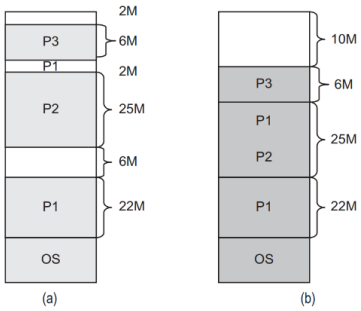
Asignación contigua con particionamiento dinámico/variable

En este caso en lugar de tener particiones estáticas, la partición de la memoria será asignada a los procesos de manera dinámica. En otras palabras, el número y tamaño de particiones no es fijo. Veamos un ejemplo para ilustrar este método.

Sean tres procesos $p_1, p_2\ y\ p_3$. Inicialmente se tiene disponible un agujero de $120MB$. p_1 consume $30MB$ de memoria desde el inicio del agujero dejando un espacio de $90MB$. De manera análoga, a $p_2\ y\ p_3$ se les asignan porciones de memoria de tamaños $40Mb\ \ y\ 48MB$ respectivamente dejando un agujero de solo $2MB$. En un siguiente momento p_2 libera le memoria utilizada dejando un agujero de $30MB$. Después llega un proceso p_4 al que se le asignan $28MB$ dejando así otro agujero de $2MB$. Enseguida p_3 libera su memoria dejando un hoyo de $48MB$. A continuación p_1 arriba de nuevo requiriendo un espacio de $30MB$. La siguiente nos permite ver este proceso paso a paso.



La fragmentación externa en el particionamiento dinámico puede ser reducido si todos los pequeños agujeros formados durante el particionamiento se compactan juntos. Esto ayuda a controlar el desperdicio de memoria ocurrido durante el particionamiento dinámico. Después de cierto tiempo el SO identifica un incremento en el número de agujeros en memoria y la compacta después de cierto tiempo, de esta manera todo ese espacio puede usarse para alojar nuevos procesos. La compactación se lleva a cabo “barajando” el contenido de la memoria de manera que toda la región ocupada se mueva hacia un solo lugar y la región desocupada hacia otro. Una limitación de la compactación es que solo puede aplicarse si la memoria se reasigna dinámicamente en tiempo de ejecución. Otra limitación es el costo ya que consume y desperdicia mucho tiempo de CPU. En la siguiente imagen se puede ver gráficamente el proceso de compactación. En esta se tiene a la izquierda una fragmentación externa con pequeños agujeros de $2\ y\ 6\ MB$, pero al compactarlos se crea un agujero de $10MB$.



Técnicas de asignación de memoria

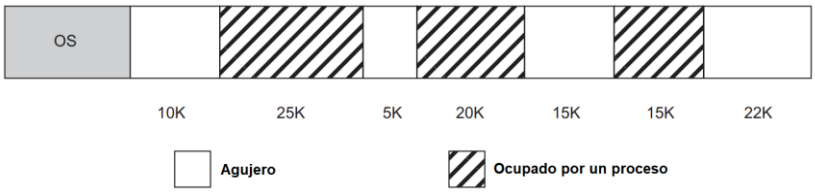
Las técnicas de asignación de memoria son algoritmos que satisfacen las necesidades de memoria de un proceso. Estas deciden cuál agujero de la lista de agujeros libres debe ser asignado al proceso en turno. Estas también se conocen como algoritmos de selección de partición. En la partición fija con particiones de igual tamaño, estos algoritmos no son aplicables precisamente porque todas las particiones tienen el mismo tamaño y por lo tanto no importa cuál de todas las particiones se elija (SON IGUALES). Por otro lado, estos algoritmos son importantes en el particionamiento fijo con particiones de diferente tamaño y en el particionamiento dinámico en términos de rendimiento de sistema y de desperdicio de memoria. Hay tres técnicas principales para la asignación de memoria.

First-fit allocation. Este algoritmo busca en la lista de agujeros libres y asigna el primero en la lista que sea lo suficientemente grande para acomodar al proceso deseado. La búsqueda se para cuando encuentra el primero que se ajuste. La siguiente vez, la búsqueda continúa en esa misma ubicación. Este algoritmo no toma en cuenta el desperdicio de memoria. Por lo que es posible que el primer agujero que encuentre sea demasiado grande comparado con el requerimiento real del proceso. Así que esto resulta en un desperdicio de memoria.

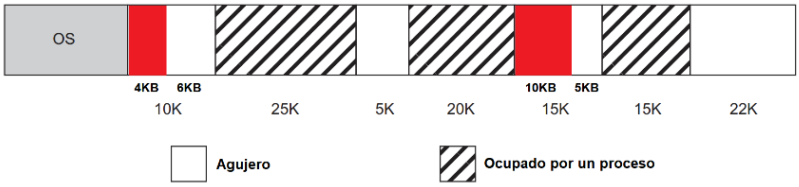
Best-fit allocation. Este compara el tamaño de memoria requerido por el proceso con los agujeros libres en la lista. Le asigna al proceso el agujero más pequeño que se ajuste al proceso. En términos de desperdicio de memoria este algoritmo es mejor que el anterior, pero existe un costo al buscar y comparar con todos los agujeros libres. Esto representa una sobrecarga adicional. Por otro lado, también deja pequeños agujeros en la memoria provocando fragmentación interna.

Worst-fit allocation. Este algoritmo es el contrario del best-fit, ya que busca en la lista el agujero más grande. Podría parecer que este algoritmo no es bueno en cuanto al gasto de memoria, sin embargo, en el particionamiento dinámico es útil ya que, debido a que los agujeros sobrantes son grandes, es posible asignarlos a otros procesos que se ajusten a ellos. Aunque al igual que el best-fit también incurre en una sobrecarga de trabajo al buscar en la lista el agujero más grande.

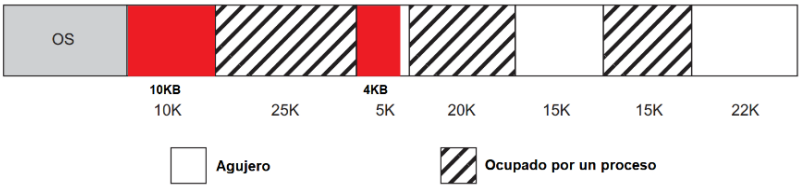
Ejemplo. Consideremos una memoria como la que se muestra en la siguiente figura y se tienen requerimientos, para asignar memoria, de $4Kb$ y $10Kb$ en ese orden. Comparar la asignación de memoria con los tres métodos anteriores.



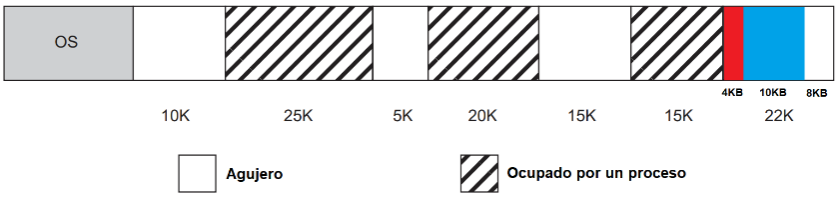
Con $first-fit\ allocation$ tenemos el siguiente escenario. Suponiendo que la búsqueda es de izquierda a derecha.



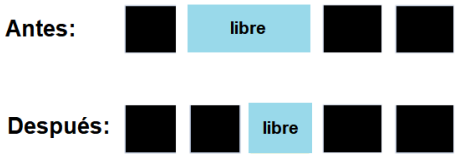
Con $best - fit\ allocation$ el espacio desperdiciado es mínimo aunque existe fragmentación interna.



Con worst-fit allocation el agujero sobrante tendrá 8KB, lo cual, también es buen tamaño para otro proceso.



Blocks splitting. En los anteriores métodos al encontrar un agujero de tamaño adecuado simplemente lo usamos, ya sea el mejor, el peor o el primero que se encuentre. Esto puede ser ineficiente si dicho bloque es mucho más grande que el proceso. Ahora la idea es dividir un agujero más grande, tomando de este un el pedazo más pequeño requerido, mientras que la otra parte quedará libre y puede ser utilizada en otras solicitudes de asignación.



Blocks coalescing. A diferencia del método anterior en lugar de dividir un agujero, se fusionan dos o tres.

Asignación de memoria no contigua

Como hemos visto la gran problemática de los métodos de asignación contigua es la fragmentación. Así que otra propuesta es la asignación de memoria no contigua. En esta los agujeros no necesitan ser contiguos, ya que pueden estar dispersos por la memoria y estar asignados a un proceso. En la asignación no contigua también tenemos particionamiento fijo y variable. Al primero se le conoce como paginación $(paging)$, y al segundo como segmentación $(segmentation)$.

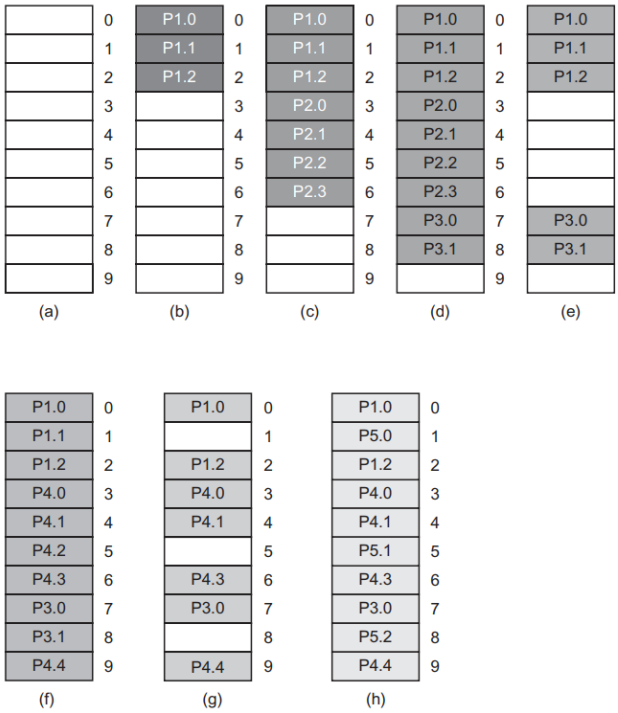
Paginación

En la paginación la memoria es dividida en particiones de igual tamaño. Cada partición tiene un tamaño relativamente más pequeño que en la asignación contigua. Estos se conocen como $marcos$ $(frames)$. Ya que la idea es reducir la fragmentacióin, la memoria lógica de un proceso se divide en pequeños bloques $(chunks)$ del mismo tamaño que los frames. Estos bloques son llamados $páginas$ del proceso. De esta manera cada vez que un frame en la memoria sea asignado a una página, esta estará bien ajustada, eliminando así el problema de la fragmentación externa. Por otro lado, el disco duro también se divide en bloques del mismo tamaño que los marcos. En pocas palabras la paginación es un concepto lógico que divide el espacio de direcciones lógico de un proceso en un conjunto de páginas de tamaño fijo, y es

implementado en la memoria física por medio de marcos. Todas las páginas de un proceso a ser ejecutado son cargadas en cualquier marco disponible en la memoria.

En el método básico de paginación, la memoria física se divide en bloques de tamaño fijo llamados $marcos$ $(frames)$, y la memoria lógica se divide en bloques también del mismo tamaño llamadas $páginas$. Cuando se va a ejecutar un proceso, sus páginas son cargadas en cualquier marco disponible desde su almacenamiento de respaldo. El almacenamiento de respaldo está dividido también en bloques de tamaño fijo de igual tamaño que los marcos de memoria.

Ejemplo. Supongamos que tenemos 10 marcos libres en memoria, y además hay cuatro procesos p_1, p_2, p_3 y p_4, los cuales, se componen de $3, 4, 2$ y 5 páginas respectivamente (Figura (a)). En la Figura (b) podemos ver que las tres páginas de p_1 han sido asignadas a sus respectivos marcos. De manera análoga todas las páginas de p_2 y p_3 se asignan como se muestra en las Figuras (c) y (d). En este punto solo queda un marco libre pero el proceso p_4 requiere de 5. Después de un tiempo, el proceso p_2 termina su ejecución y libera su memoria: desde el marco 3 al 6. Por lo tanto, se tienen 5 marcos libres como se ve en la Figura (e). Ya que el proceso p_4 requería 5 marcos, entonces las cuatro primeras páginas de este proceso se asignan a los marcos 3 hasta el 6, y el último marco se asigna a la quinta página de p_4, ver Figura (f). Supongamos ahora que después de otro tiempo $p_1$libera su página 1, p_4 libera su página 2, y p_3 libera su página 1 como se muestra en la Figura (g). Dichas páginas han sido enviadas al disco duro (swapped). En este punto un proceso p_5 llega al sistema necesitando tres páginas. Estas pueden ser asignadas en memoria, como se muestra en la Figura (h), en tres marcos NO contiguos.



En general para cargar un programa, se debe comenzar desde el inicio y recoger las páginas una a una. Cada página puede ser cargada en cualquier marco disponible. Antes de comenzar el proceso de carga se realiza un proceso de verificación para determinar si existes suficientes marcos o no. Una variable dentro del sistema operativo puede llevar la cuenta de esto de manera que se pueda verificar en cualquier momento el número de marcos disponibles. Cada vez que la computadora es iniciada esta variable contiene el número total de marcos disponibles en memoria principal. Después cada vez que una página es cargada en la memoria principal se disminuye en uno esa variable contador, y cada que se remueve una página se incrementa en uno.

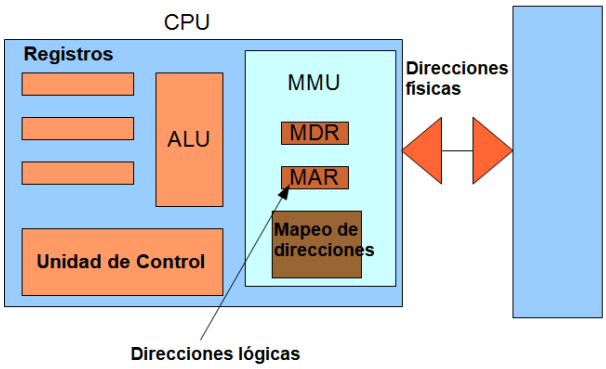
A medida que se carga cada página, el número de marco correspondiente se almacena en una tabla llamada $tabla\ de\ páginas$. Cada entrada de la tabla es una estructura bidimensional cuyo campo principal es el número de marco, aunque se pueden incluir otros campos que almacenen más información. El número de página es usado como un índice dentro de la tabla. Así cada proceso tiene su propia tabla de páginas. Por ejemplo la siguiente tabla indica que las cuatro páginas de lontud para un programa se han cargado en los marcos $4,12,13\ y\ 46$ respectivamente.

Número de página	Número de marco
0	4
1	12
2	13
3	46

Un aspecto importante de la páginación es la $traducción\ de\ direcciones$. Para cargar un programa en la memoria, usando paginación, no se lleva a cabo ninguna modificación de la dirección. Por ejemplo, si la dirección lógica es cero, el contador del programa se fija en cero para comenzar la ejecución del programa. En este caso será obvio que la probabilidad de que la página cero sea cargada en el marco cero es prácticamente nula. Así que la ubicación actual de dicha instrucción depende del número de marco donde se haya cargado esa página. El proceso de calcular la dirección en memoria principal de una dirección lógica es llamado $traducción\ de\ direcciones$.

Traducir las direcciones es necesario para todas las direcciones de instrucción y todas las direcciones de operando que no son absolutas. Hay al menos una traducción de dirección para cada instrucción de máquina (para cada dirección de instrucción), y se lleva a cabo antes de extraer $(fetched)$ la instrucción. El número total de traducciones de dirección para cada instrucción es uno más el número de operandos no absolutos.

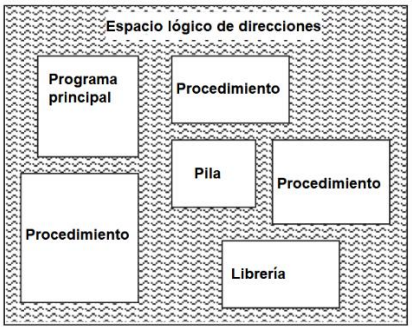
La traducción debe ser muy rápida para evitar que la ejecución de un programa sea lenta, y para ello dicha traducción debe ser realizara por hardware siempre que sea posible. El módulo que lleva a cabo la traducción de dirección es llamado Unidad de Administración de Memoria (Memory Management Unit – MMU). La entrada a la MMU es una dirección lógica y la salida es la correspondiente dirección física. Esta dirección es entonces enviada al registro de $direcciones\ de\ memoria\ (MAR)$ para que pueda escribirse o leerse sobre dicha ubicación.



La paginación es más eficiente que los métodos de asignación contigua ya que no necesita realizar una compresión periódica en la memoria principal, así que no consume ese tiempo. Por otro lado, tiene las desventajas de invertir tiempo en la traducción de dirección, y de requerir espacio para almacenar las tablas de páginas de todos los procesos del sistema.

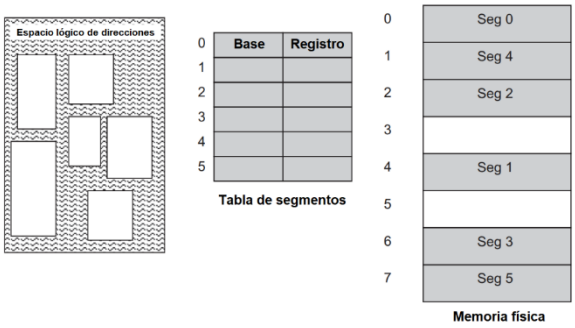
Segmentación

En general un programador escribe programas en términos de módulos (no de páginas). Esto reduce la complejidad del problema a una serie de módulos: programa principal, procedimientos, pila, datos, etc. La segmentación es una técnica de administración de memoria que soporta el concepto de módulos. En esta técnica los módulos son llamados segmentos. Los segmentos son divisiones lógicas de un programa y pueden tener diferente tamaño. En el caso de la paginación, las páginas tienen igual tamaño.



La division del espacio lógico de direcciones en segmentos de diferente tamaño elimina el problema de la fragmentación interna ya que cada segmento se ajustará al espacio requerido. Cada segmento está identificado por su nombre y su longitud. Por otro lado la dirección lógica se divide en dos partes: el nombre del segmento y su desplazamiento (la ubicación de memoria) dentro de ese segmento. Para cada programa ejecutable hay tres segmentos principales: segmento de código, segmento de datos y segmento de pila. Cada uno de estos segmentos puede hacer uso de los otros.

Para convertir la dirección lógica en la dirección física, la dirección inicial del segmento en la memoria debe ser conocida, es decir, el registro base.



Memoria virtual

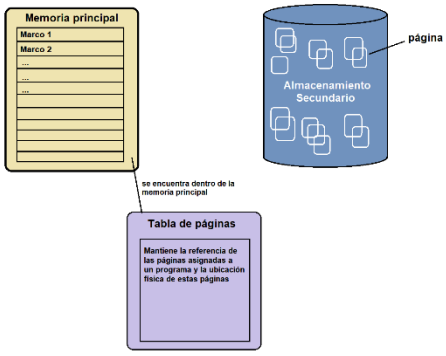
Los algoritmos para el manejo de memoria como los anteriores son necesarios ya que las instrucciones del proceso en ejecución deben estar en la memoria física. Sin embargo, estos tienen diferentes desventajas como las siguientes:

- Hay partes de la memoria principal que se quedan sin uso (fragmentación externa), o incluso páginas completas también se pueden quedar sin uso si el programa es muy pequeño.
- El tamaño máximo del programa y de sus datos está limitado por el uso de la memoria principal. Aquellos programas que sean más grandes jamás se ejecutarán.
- Cuando un programa entra en la memoria principal, permanece ahí hasta que termine su ejecución incluso si solo una parte del programa o de sus datos se hayan utilizado.

Para administrar la memoria de maneja más eficiente y con menos errores, los sistemas modernos proporcionan una abstracción de la memoria principal conocida como $Memoria\ Virtual\ (MV)$. Esta se refiere al conjunto de mecanismos habilitados por el sistema operativo para simular, de forma transparente para el usuario, la existencia de mayor cantidad de memoria física que la que está instalada en realidad. Si el tamaño combinado del programa, sus datos, y la pila exceden la cantidad de memoria física disponible para almacenarlos, el sistema operativo mantiene solamente las partes activas del programa en la memoria principal mientras que el resto en el disco duro. De esta manera se puede ejecutar un proceso sin estar cargado completamente en la memoria. Esta proporciona tres capacidades importantes.

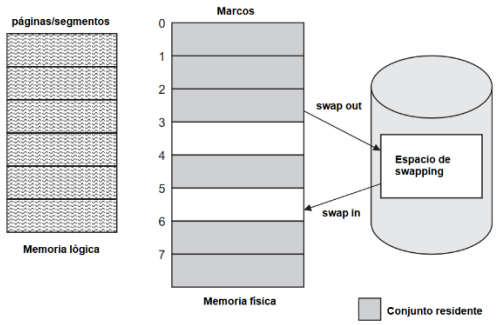
1. Usar la memoria de manera más eficiente, manteniendo en memoria solo las áreas activas, transfiriendo los datos entre el almacenamiento secundario y la memoria según sea necesario.
2. Simplifica la administración de memoria proporcionando a cada proceso un espacio de direcciones uniforme.
3. Protege de corrupción el espacio de direcciones de cada proceso.

La administración de memoria virtual puede llevarse a cabo utilizando las mismas técnicas de paginación y segmentación vistas anteriormente. En general el esquema de MV se compone de la MMU (Memoria de Administración de Memoria) que se encuentra en el procesador, el búfer de traducción de adelantada o TLB que es una memoria de tipo caché administrada por la MMU y que contiene partes de la tabla de paginación, la memoria principal y el almacenamiento secundario.



Para que un programa pueda ejecutarse sin estar completamente cargado en memoria se lleva a cabo la paginación, que como hemos visto anteriormente divide la memoria principal en pequeños trozos del mismo tamaño llamados marcos, por otro lado los procesos también se dividen en pequeños trozos del mismo tamaño llamados páginas. Así, en cada instante solo residen en memoria una fracción de páginas de un proceso. Estas páginas que están en memoria se administran por medio de la tabla de páginas, que se encuentra dentro de la memoria principal. Esta tabla mantiene una referencia de las páginas asignadas a un programa con la ubicación física de estas.

Carga por demanda de componentes de proceso. En general sería un desperdicio de memoria si se cargan todos los componentes de un proceso en la memoria, ya que cualquier proceso puede ser ejecutado sin cargar todos sus componentes. Una de las ventajas de la MV es que solo aquellos componentes requeridos son cargados primero en la memoria mientras que los otros solo son cargados cada que se requieren. La regla es: ***nunca cargar un componente de un proceso a menos que sea necesario.*** Los componentes de un proceso que están presentes en la memoria se conoce como $conjunto\ residente$.



La ejecución de un proceso se lleva a cabo sin problemas siempre y cuando la dirección lógica generada por el procesador esté en el conjunto residente del proceso. Es decir, que todos los componentes requeridos del proceso estén en la memoria. Sin embargo, puede suceder que cuando un componente (una página o un segmento) correspondiente a una dirección lógica generada no se encuentre en el conjunto residente de ese proceso. Así que para ejecutar dicho proceso, los componentes que no están en memoria necesitan ser cargados ahí. Estos componentes se encuentran en el disco duro en un área separada llamada $espacio\ de\ intercambio$. Si el procesador genera una direccio´n lógica que no es encontrada en memoria después de la traducción de dirección, entonces se genera una interrupción de acceso a memoria. Esto significa que un componentes corerspondiente a la memoria lógica generada no está en la memoria en ese momento. El proceso involucrado es entonces bloqueado por el sistema operativo, y para que continúe su ejecución los componentes necesarios necesitan pasar del disco duro a la memoria: $swap\ in$. Después el sistema operativo cambia el estado del proceso de bloqueado a listo para volver a ejecutarse.

Algunos problemas a resolver en la implementación de carga por demanda en un sistema de MV son:

i. ¿Cómo reconocer cuál componente está en memoria y cuál no?
ii. ¿Cuántos procesos podrán residir en la memoria.
iii. ¿Cuánta memoria se le asignará a cada proceso?
iv. Cuando un componente es requerido del disco duro para ser llevado a la memoria, es posible que no haya marcos libres. ¿Dónde serán almacenados esos componentes entonces? Aquí la idea es reemplazar algún componente que ya esté almacenado y hacer espacio para el nuevo componente. Así, un componente será llevado al disco duro $(swapped\ out)$ y el nuevo componente llevado a memoria $(swapped\ in)$. A este proceso se le conoce como remplazo de componentes. Esto nos lleva a preguntarnos ¿Cuál será la estrategia para reemplazar un componente? Estas estrategias son conocidas como algoritmos de reemplazo de componentes.
v. La MV utiliza espacio del disco duro como memoria, y con ayuda de esto la MV es capaz de administrar procesos de gran tamaño o múltiples procesos. Este espacio en el disco duro se conoce como espacio de intercambio $(swapping\ space)$. Este espacio debe ser administrado eficientemente para que la MV funcione sin problemas.

Paginación en Memoria Virtual

Todos los conceptos y detalles que se aplican para la paginación usando solo la memoria real también se aplican a la memoria virtual. Es decir, cada programa se divide lógicamente en piezas de igual tamaño llamadas $páginas$, y por otro lado la memoria se divide en piezas también del mismo tamaño llamados $marcos$. Además el tamaño un frame y una página son iguales. Para diferenciarlas, la paginación sobre memoria virtual es llamada $demanda\ de\ páginas\ (demand\ paging)$. Al igual que la carga por demanda de componentes, en la paginación con MV solo se cargan las páginas en el instante en que son requeridas.Por otro lado, cuando se realiza $swapping$ no se almacena en disco (ni se devuelve a memoria) el proceso completo, ya que no es necesario tener todo el proceso en memoria para poder ejecutarlo. A esto se le suele llamar lazy $swapping$. Las operaciones de intercambio también son conocidas como $page-in$ y $page-out$.

La mayor diferencia de paginación en memoria virtual se da en la carga de un programa a memoria. Usando memoria virtual, para que un programa comience a ejecutarse no es necesario cargarlo completamente en la memoria, mientras que en la paginación normal sí.

En los modelos de memoria virtual, la memoria principal se considera un lugar temporal para un proceso o parte de un proceso. Una página que es llevada a memoria puede ser cambiada si momentos después se decide que debe ser removida. Para no perder los cambios hechos, la página es copiada a su lugar en el disco duro. Así que el disco duro es el lugar donde se mantiene la imagen completa de un proceso, mientras que la memoria solo es un alojamiento temporal. Una página en el disco duro y su copia en memoria no necesariamente deben ser iguales todo el tiempo, ya que la copia refleja los últimos cambios realizados en el contenido de dicha página.

En teoría una memoria virtual muy pequeña, administrada con paginación, es adecuata para ejecutar un programa bastante grande y para implementar multiprogramación. De hecho podría componerse solo de unos cuantos marcos. Cuando el sistema quiere ejecutar una instrucción, esta y sus operandos deben estar presentes en la memoria principal. De lo contrario, ocurre un $\boldsymbol{error\ de\ página}$ y el proceso es suspendido hasta que la página requerida sea llevada a memoria. Si se necesitan más páginas, y no están presentes en memoria, entonces más de un error de página ocurrirán para esa instrucción. Cada vez que ocurre una falla, el proceso se suspende y entonces se puede seleccionar otro proceso para usar el CPU.

Si se da el caso que el procesador pasa la mayor parte del tiempo suspendiendo y despertando procesos debido a frecuentes errores de páginas entonces se da una condición llamada $thrashing$. Dicha situación debe ser evitada. Hay que tomar en cuenta que la probabilidad de que ocurra $thrashing$ es más alta cuando la memoria principal no es suficientemente grande para el número de procesos o programas activos.

Una referencia a una dirección lógica que ya ha sido cargada en la memoria principal se denomina $page\ success$ o $página\ exitosa$. Pero si dicha referencia se hace y la página correspondiente no está en memoria ocurre un $fault\ page$ o error de página.

La administración por paginación de memoria virtual no trabaja bien para procesos que están diseñados como un simple módulo con demasiados saltos ($go\ to$). Afortunadamente, los programas actuales están escritos de manera estructurada, modular y orientado a objetos. En estos cuando un bloque o módulo de un programa está siendo ejecutado, hace referencia a un pequeño número de páginas. A esto se le llama localidad de referencia (locality of reference), y significa que no todas las páginas de un proceso deben cargarse en memoria en todo momento. El proceso seguirá trabajando sobre una pequeña parte del programa. Es decir, la ejecución de un módulo puede excluir la ejecución de otros. Por ejemplo,si se invoca un módulo de la cláusula $then$ en la estructura $if - then - else$, es poco probable que aparezca en la cláusula $else$ y viceversa.

Hay tres técnicas fundamentales para hacer lugar para una página entrante: (1) mandar a disco (swap-out) una página de un proceso, (2) mandar a página un segmento de un proceso y (3) mandar a disco un proceso completo. Tomando en cuenta estas tres alternativas los algoritmos para administracióin de memoria pueden usar dos o tres de estas.

Algoritmos de reemplazo de página

La idea central de la memoria virtual es eliminar el tamaño de la memoria principal como un parámetro que restringe el número de procesos que pueden procesarse en el sistema. Con la memoria virtual podemos tener más procesos en el sistema que si solo usamos la memoria real. Como hemos visto la paginación es una forma de tener más procesos si solo almacenamos en memoria las partes de cada proceso que son necesarias. Para ello se cuenta con dos operaciones esenciales: $swap - out$ que también es llamada $page - ou$t o $page\ removal,$ y $swap - in$ conocido también como $page - in$ o $page - swapping$. Los algoritmos de reemplazo de página son aquellos que realizan estas operaciones de manera que el rendimiento del sistema sea lo más cercano al óptimo.

Hay que recordar que un programa, en su totalidad, se guarda en un dispositivo de almacenamiento secundario, y solo se llevan copias temporales de algunas páginas a la memoria principal. Una página en memoria puede contener instrucciones y/o datos, y cualquiera de estos puede modificarse durante su permanencia en la memoria principal. Así que cuando una página ha sido modificada ya no es una copia exacta de su correspondiente página en el disco duro. Por lo tanto, si una página es removida de la memoria, entonces debe ser copiada otra vez en el disco duro. De no hacer esto los resultados de la ejecución no se reflejarán en la transcripción original del programa. Por una cuestión de rendimiento, una página que no ha sido modificada no necesita copiarse otra vez en el disco duro.

Para distinguir entre una página modificada de una no-modificada se usa un bit bandera llamado $bit\ modificado\ (modified\ bit)$, el cual es denotado M. Este bit se agrega a cada entrada de cada tabla de páginas o cada marco de la memoria física dependiendo de la arquitectura del hardware o del algoritmo de reemplazo de página. El bit M se fija siempre que se almacena algo en el marco de la página correspondiente. Su valor inicial será cero cuando la página es almacenada en memoria desde el disco duro. Una página cuyo bit M se fija en 1 es copiado de vuelta a su lugar original siempre que sea removido de memoria, pero si el bit es 0 entonces no se copiará.

Política FIFO. Uno de los métodos más simples de remplazo de pagína es el método $FIFO$ $(First\ In\ First\ Out)$. Cuando se desea remover una página de la memoria principal, aquella que haya estado en memoria más tiempo será removida. Para llevar un seguimiento del tiempo que pasa cada página en memoria, cada que se realice $page - in$ sobre alguna página se le adjuntará un sello de tiempo $(timestamp)$. Dicho número puede ser el tiempo exacto que ha pasado cada página en memoria o un número que solo muestra el orden en que fueron cargadas las páginas. Así, aquella página con el sello de tiempo más bajo será elegido.

La implementación no contempla la búsqueda, en todo el conjunto, de la página más vieja en memoria ya que esto sería un gasto de tiempo considerable. Sin embargo, es posible mantener una lista doblemente ligada con dos apuntadores: al frente y al final de la lista. Al cargar una página en memoria, se creará un nodo con la informacióin de la página y dicho nodo se agregará al final de la lista. Y al remover una página se tomará en cuenta la información contenida en el nodo del frente de la lista ligada. Cuando la página elegida se remueva de la memoria también el nodo será eliminado de la lista.

Este método es rápido, sin embargo, una desventaja es el espacio extra requerido para almacenar la lista doblemente ligada. Además de esta, otra desventaja importante de este método es la llamada $anomalía\ de\ Belady$. En este se indica que es posible tener más errores de página al aumentar el número de marcos utilizando el método FIFO.

Ejemplo. Las peticiones de página o (referencia de página o cadena de referencia de página) es el conjunto ordenado de números de páginas que se genera cuando un programa es ejecutado.

Peticiones de página	1	2	3	4	1	2	5	1	2	3	4	5
Marco 0	1	1	1	4	4	4	5	5	5	5	5	5
Marco 1		2	2	2	1	1	1	1	1	3	3	3
Marco 2			3	3	3	2	2	2	2	2	4	4
Fallo o éxito	x	x	x	x	x	x	x	e	e	x	x	e

Peticiones de página	1	2	3	4	1	2	5	1	2	3	4	5
Marco 0	1	1	1	1	1	1	5	5	5	5	4	4
Marco 1		2	2	2	2	2	2	1	1	1	1	5
Marco 2			3	3	3	3	3	3	2	2	2	2
Marco 3				4	4	4	4	4	4	3	3	3
Falla o éxito	x	x	x	x	e	e	x	x	x	x	x	x

Otro ejemplo: 15 fallas con 3 marcos

Peticiones	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Marco 0	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
Marco 1		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
Marco 2			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
¿Falla?	x	x	x	x	-	x	x	x	x	x	x	-	-	x	x	-	-	x	x	x

10 fallas con 4 marcos

Peticiones	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Marco 0	7	7	7	7	7	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2
Marco 1		0	0	0	0	0	0	4	4	4	4	4	4	4	4	4	4	7	7	7
Marco 2			1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
Marco 3				2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1
¿Falla?	x	x	x	x	-	x	-	x	-	-	x	-	-	x	x	-	-	x	-	-

Peticiones	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Marco 0	7	7	7	7	7	7	7	4	4	4	4	4	4	4	4	4	4	4	4	4
Marco 1		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	7	7
Marco 2			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
Marco 3				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1
Marco 4						3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
¿Falla?	x	x	x	x	-	x	-	x	-	-	-	-	-	-	-	-	-	x	x	x

Reemplazo óptimo. En este algoritmo la política de reemplazo va de acuerdo a cuál página no será referenciada en un tiempo más largo.

Con FIFO tedremos 9 fallas con 3 marcos

Peticiones	5	0	2	1	3	4	2	1	5
Marco 0	5	5	5	1	1	1	2	2	2
Marco 1		0	0	0	3	3	3	1	1
Marco 2			2	2	2	4	4	4	5
¿Fallo?	x	x	x	x	x	x	x	x	x

Con reemplazo óptimo tendremos 7 fallas con 3 marcos. El desarrollo de este es como sigue: para las tres primeras peticiones de las páginas 5, 0 y 2 respectivamente hay tres errores de página ya que no se encontraban las correspondientes referencias. En este, que es el tiempo 3, vemos que para la página 5 la primera referencia hacia adelante está a 6 unidades de tiempo, para la página 0 ya no hay referencia hacia adelante entonces podemos tomarla como un tiempo infinito. Finalmente, para la página 2 (aún en el tiempo 3) la referencia hacia adelante está a 4 unidades de tiempo. Por lo tanto, la página que debería salir en el siguiente intercambio es 0, lo cual denotamos con color rojo.

En el tiempo 4 la petición es para la página 1, como ya no se encuentra esa referencia en la memoria, entonces provoca un fallo, y se intercambia con el 0 que se había elegido como la página a intercambiar. Ahora, antes de pasar al tiempo 5, primero debemos decidir cuál será la página a intercambiar, en este caso para la página 5 su referencia hacia adelante está a 5 unidades de tiempo. La referencia de 1 está a 4 unidades, y la de 2 está a 3 unidades, por ello elegimos la página 5 (en rojo) como la página que debe salir. Este proceso se sigue una y otra vez hasta obtener la tabla como la que se muestra a continuación.

Tiempos	1	2	3	4	5	6	7	8	9
Peticiones	5	0	2	1	3	4	2	1	5
Marco 0	5	5	5	5	3	4	4	4	5
Marco 1		0	0	1	1	1	1	1	1
Marco 2			2	2	2	2	2	2	2
¿Fallo?	x	x	x	x	x	x	-	-	x

Otro ejemplo. Con FIFO 9 fallas para 3 marcos

Peticiones	2	3	4	1	5	2	3	5	4	2	1
Marco 0	2	2	2	1	1	1	3	3	3	3	3
Marco 1		3	3	3	5	5	5	5	5	2	2
Marco 2			4	4	4	4	4	4	4	4	1
¿Falla?	x	x	x	x	x	x	x	-	-	x	x

Con algoritmo óptimo 7 fallas con 3 marcos

Tiempos	1	2	3	4	5	6	7	8	9	10	11
Peticiones	2	3	4	1	5	2	3	5	4	2	1
Marco 0	2	2	2	2	2	2	2	2	2	2	2
Marco 1		3	3	3	3	3	3	3	4	4	4
Marco 2			4	1	5	5	5	5	5	5	1
¿Falla?	x	x	x	x	x	-	-	-	x	-	x

Least Recently Used Policy (LRU). En este algoritmo se elige, para reemplazar a la página que más tiempo lleve sin ser usada. En el ejemplo que se muestra en la siguiente tabla podemos ver, en el tiempo 5 que la petición para la página 1 está a solo una unidad de tiempo a la izquierda, para la página 4 es cero ya que acaba de ser accedida en el paso anterior, y para la página 2 su referencia más cercana está a dos unidades de tiempo a la izquierda, por lo tanto, la página 2 es la elegida (denotada en rojo) para ser reemplazada. En cada tiempo podemos ver en rojo la página que tiene más tiempo (más peticiones hacia la izquierda) en haber sido accedida, y esa página es precisamente la que se reemplaza en la siguiente unidad de tiempo.

Tiempos	1	2	3	4	5	6	7	8	9	10	11	12
Peticiones	4	3	2	1	4	3	5	4	3	2	1	5
Marco 0	4	4	4	1	1	1	5	5	5	2	2	2
Marco 1		3	3	3	4	4	4	4	4	4	1	1
Marco 2			2	2	2	3	3	3	3	3	3	5
¿Falla?	x	x	x	x	x	x	x	-	-	x	x	-